

**DISPATCHER CONFIGURABLE LINKING OF SOFTWARE ELEMENTS****CROSS REFERENCE TO RELATED APPLICATIONS**

5 This application claims priority from U.S. Provisional Application Serial No. 60/192,275, filed March 27, 2000, which is hereby incorporated by reference in its entirety as if fully set forth herein.

**BACKGROUND OF THE INVENTION****FIELD OF THE INVENTION**

10 The present invention relates to telecommunication systems and, particularly, to a system and method for linking software elements in a telecommunications system.

**DESCRIPTION OF THE RELATED ART**

15 Telephony systems are becoming increasingly complex. The widespread use of the Internet Protocol in data communications systems has led to interest in "Voice over IP" (VoIP) and "Telephony over LAN" (ToL) applications. In particular, several IP telephony protocols have been developed, including the H.323 Recommendation suite of protocols  
20 promulgated by the International Telecommunications Union (ITU), the Session Initiation Protocol (SIP), and Media Gateway Control Protocol (MGCP), to name a few.

At the same time, there remains a large installed base of users of traditional private branch exchange (PBX) networks. While such users would  
25 benefit from low cost IP telephony, the idea of replacing such existing systems may be undesirable. An intermediate solution is the use of a Telephony Internet Server that interfaces a PBX and a packet network, and provides signaling conversion between protocols used by the PBX and by the packet network.

30 To be viable, such Telephony Internet Servers need to be able to dynamically add features. Moreover, it is desirable to balance system workload.

005727 95927460

### SUMMARY OF THE INVENTION

These and other problems in the prior art are overcome in large part by  
5 a system and method according to the present invention.

A telecommunications system having a software dispatcher is provided for delivering messages between dispatcher clients, i.e., software subsystems that may be in the same process, a different process, or on a different machine. The dispatcher manages a pool of threads to balance the  
10 workload. The dispatcher can process both synchronous and asynchronous messages by dispatching the message to all registered subsystems in order of their registered priority.

The dispatcher thus has the ability to dynamically add features to telephony systems. The features can run in the workspace of the original  
15 application or be in another process. The separate process can run on the same system as the original application or another system on the same network.

### BRIEF DESCRIPTION OF THE DRAWINGS

20 A better understanding of the invention is obtained when the following detailed description is considered in conjunction with the following drawings in which:

FIG. 1 is a diagram illustrating a network employing a dispatcher according to an implementation of the present invention;

25 FIG. 2 is a diagram illustrating message lists according to an implementation of the invention;

FIG. 3 is a diagram schematically illustrating message registration according to an implementation of the invention;

FIG. 4 is a diagram illustrating message base structure according to an  
30 implementation of the present invention; and

FIG. 5 is a diagram illustrating message queues according to an

006TAT 06924260

implementation of the invention.

Fig. 6 depicts an object tree in accordance with the present invention.

### DETAILED DESCRIPTION OF THE INVENTION

5           FIGS. 1- 5 illustrate an improved system and method for message handling in a software system and, particularly, in a communications system. A dispatcher is provided for delivering messages between dispatcher clients, i.e., software subsystems that may be in the same process, a different process, or on a different machine. The dispatcher manages a pool of  
10 threads to balance the workload. The dispatcher can process both synchronous and asynchronous messages by dispatching the message to all registered subsystems in order of their registered priority.

Turning now to FIG. 1, a telecommunications network according to a particular implementation of the invention is illustrated therein and generally  
15 identified by the reference numeral 100. The telecommunications network 100 includes a switch, such as a private branch exchange (PBX) 106. The PBX 106 may be embodied as the Hicom 300, available from Siemens Information and Communication Networks, Inc., Boca Raton, Florida. The PBX 106 couples to the public switched telephone network (PSTN)(not  
20 shown) and a plurality of user telephony devices 108a-108n, such as telephones, facsimile machines, and the like. A packet network server or telephony Internet server 104 according to an implementation of the invention intercouple the PBX 106 to a packet network 102, such as the Internet, a corporate Intranet, or a local area network (LAN).

25           The packet network server 104 may be linked to the PBX 106 over a T1/E1 interface and communicate using any of a variety of protocols, such as E&M Wink Start, AT&T 4ESS SDN, CorNet-N, or CorNet NQ. The packet network server 104 communicates with the packet network using any of a variety of packet telephony protocols, such as H.323 or the Session Initiation  
30 Protocol (SIP). The packet network server 104 may be embodied as the Hicom Xpress Telephony Internet Sever (TIS) 2.0 or 2.1, implemented on a

006742696-121900

Windows NT platform, available from Siemens Information and Communication Networks, Inc., Boca Raton, Florida.

5 The packet network server 104 implements a controller, referred to as the dispatcher 107, for linking software modules. The dispatcher 107 handles the delivery of messages to subsystems which have registered to receive particular messages. A subsystem to which messages may be delivered can be internal or external to the application which contain the dispatcher 107. That is, subsystems of external applications may register with an application's dispatcher subsystem to receive messages. Such subsystems can include, 10 for example, call processing subsystems and device handler subsystems. It is noted that, while described herein focusing primarily on telecommunications system software in a particular architecture, the invention is not so limited. Thus, the dispatcher 107 may be implemented as software for any complex system.

15 As noted above, the dispatcher 107 maintains a list of all messages in the system. The messages are identified by a unique integer and a node (MsgRcvList) in the dispatcher. Each party interested in receiving the message constructs a node (called a MessageReceiver) that is linked into the MsgRcvList. The list of message receivers is ordered via a numeric priority 20 value. When a subsystem has a message it wants to deliver, it informs the dispatcher 107 of which message is to be delivered. The subsystem is unaware of the destination of that message. Only the dispatcher 107 knows what the destination will be. Thus, when a subsystem registers with the dispatcher 107, it is really telling the dispatcher 107 what message(s) it is 25 interested in processing and which it is capable of sending. The dispatcher 107 has no knowledge as to the semantics of a message or the contents (attached parameters) of a message. It only knows a message as a unique integer and uses this integer to identify the list of subsystems which are to receive the message. Meaning and content of a given message is relevant 30 only to the particular subsystems which either issue or receive the message.

As shown in FIG. 2, the dispatcher 107 maintains a list 200 of

006727" 96924260

5

10

15

20

30

delivered a message identified by *MessageId* if the *MessageSubId* in the message also matched. *Priority* allows the message to be registered for with a specific priority with respect to other subsystems interested in the same message. For instance, a subsystem performing a background trace

5 operation of messages would register for the messages at a lower priority than a subsystem doing some time critical processing of the message. Using this information, the dispatcher 107 can update the message receiver list (*MsgRecvList*) for the given message (if the list is already existing) by placing a *MessageReceiver* entry for the subsystem at the appropriate

10 position in the linked list of message receivers, based on priority. If a *MsgRecvList* does not yet exist for the message, a new *MsgRecvList* is created.

While processing a dispatched message, a message receiver or subsystem will read the incoming message and its parameters, as will be

15 explained in greater detail below. Depending on its contents, the message receiver may choose to perform actions such as dispatching a new message, adding or altering the parameters of the current message, or choose to do nothing with the message. The message receiver can tell the dispatcher 107 to continue dispatching the message or prevent any message receiver with a

20 lower priority from seeing the message. These allow new message receivers to view dispatched messages and change the behavior of the message flow.

If an error is detected while processing the message, the reporting of the error is the responsibility of the message receiver processing the message. This distributes the error handling across all of the message

25 receivers rather than have the sender know what can go wrong at any stage of processing the message. Errors like an invalid parameter, having a required parameter missing or having another API call fail can cause the processing message receiver to dispatch an error message to a message receiver that is part of the dispatcher subsystem and/or set the return code of

30 the message to indicate the failure.

The messages themselves are structured in Flexible Message

Parameters (called *MsgBase*), because dynamic extensions to the dispatcher "network" may require additional parameters or information. The *MsgBase* is a dynamic data structure defined by a collection of 3-tuple fields: *name*, *type*, and *value*. The *name* field is an ASCII string to minimize the possibility of collision with other dynamically added fields. The *type* field is an enumeration that describes the format of the data in the *value* field. The *value* field contains (or points to) the actual parameter data, as shown in FIG. 4. Because the dispatcher 107 knows all types of data within the *MsgBase*, the transfer of a *MsgBase* across a network (or other medium, including persistent storage) is possible regardless of the layout and types of data in the *MsgBase*. Because one possible type for a field is another *MsgBase*, complex data structures can be managed. *MsgBase* objects can be dispatched locally or transported across the network transparently via a common API (application programming interface).

For example, FIG. 4 illustrates several exemplary flexible message parameters 400, each including a name 402, a type 404, and a value 406. Thus, parameter 400a has the name "B Channel", a type ULONG, and a value of 17; parameter 400b has a name of CP number, a type of STRING, and a value of 1234, and the parameter 400d has a name of state, a type ULONG, and a value 4. Further, in the example shown, the parameter 400c has a name STATE, a type ULONG, and a value of other flexible message parameters 400e or 400f.

As noted above, the dispatcher 107 can process messages in both a synchronous and an asynchronous manner. A command referred to as the *sendMessage* command applies to synchronous processing, and a command referred to as the *postMessage* command applies to asynchronous processing.

A message that is sent via *sendMessage* is immediately processed serially by each subsystem registered to receive the message on the sender's thread of execution. The *sendMessage* call will not return to the caller until all receivers have processed the message. The *sendMessage* method of the

5       A message is sent via *postMessage* when the sender does not want or  
need to wait for the processing of the message to complete. The  
*postMessage* method of the dispatcher allows a subsystem to issue a  
message to be processed in a different context, thus allowing the issuing  
subsystem to complete any remaining processing in its own context prior to  
10 the issued message being processed by other subsystems.

30        The number of threads actually managed by the dispatcher 107 is  
dynamic, based on load with some defined minimum and maximum number



of threads. The dispatcher 107 can add or remove threads as the load dictates. Operating system semaphores are used to keep the dispatcher's data members thread safe while adding and deleting from the pool of threads.

As messages are being dispatched, there is potential for a subsystem's code  
5 to process a message to be executed from different threads at the same time.

If the subsystem's code to handle messages is non-reentrant, it may register with the dispatcher to use a semaphore when accessing the code to process messages and thus make the code thread safe.

In a telephony system, *PostMessage* is used to pass messages from a  
10 device handler subsystem (DH)(not shown) to a call processing subsystem (CP)(not shown). The device handler recognizes messages arriving asynchronously from external devices, such as a telephone going off hook or an ISDN layer 2 message, but is not capable of processing the message in the context of a call. It is CP's job to process the message. DH and CP each  
15 operate within one or more threads. The task of DH is usually small and well defined: detect a message, package it and pass it on. CP's task, on the other hand, may be quite complex depending on things like the number of features interested in a message.

Since DH could conceivably be flooded with messages for periods of  
20 time, it is not desirable to do CP's processing in the context of DH (which would happen if DH delivered its messages to CP via *SendMessage*). Therefore, DH delivers its messages to CP (and any other interested subsystems) via *PostMessage*. This concept is especially beneficial in a telephony system where an LMQ can be assigned for each call. Thus, the  
25 number of threads in use is based on the number of active calls (with messages to be processed) and the available processor time can be evenly (more or less) distributed across the current call load of the system.

In one implementation, the dispatcher and its clients are maintained in an object oriented system. As can be appreciated, in such a system, a large  
30 number of objects must be maintained. To help organize the system, the Dispatcher, in one implementation, provides an "object tree." It is noted that,

006121 960942 09

In any event, an object tree is a hierarchical collection of named objects (similar to a UNIX file system). These objects can be inserted, referenced and deleted just as with a file system. As shown in FIG. 6, the object tree is populated by container objects 604 and base objects 602. A container object 604 can have other container or base objects 606, 608 under it. A base object is one that has a specific purpose and cannot be further subdivided.

- **Hashed, in-memory:** This container simply contains other objects and indexes them by a string. This is similar to a directory file in UNIX (although in memory, not on disk).
- **Array, in-memory:** Some objects (such as subscribers) are best described by their sequential characteristics. The Array container accepts array subscripts instead of names (as with the Hashed, in-memory container).

The object tree is used as an API point for several system components. The database for example, attaches itself to the object name tree and appears like a container object. All objects that are placed in this

container persist beyond restarts of the gateway software.

The invention described in the above detailed description is not intended to be limited to the specific form set forth herein, but is intended to cover such alternatives, modifications and equivalents as can reasonably be included within the spirit and scope of the appended claims.

5

10

006T2T" 96924/260